Exam solution for Advanced Programming 2013

Niels G. W. Serup

December 17, 2013

Contents

1 The Salsa Language

All Salsa-related source files are in `src/salsa`.

1.1 Grammar modification

Operator associativity is left unspecified in the grammar. I have chosen to assume left associativity where the choice between left and right affect the semantics, i.e. in `Expr '-' Prim`, and accept the ambiguity where the semantics are unaffected, i.e. in `Expr '+' Prim` and `Command '||' Command`.

I keep it ambigious because I think only the semantics of parsed programs are important.

Specifically, with `Expr '-' Prim` you can have `2 - (1 - 1) != (2 - 1) - 1`, but the same is not the case for `+` and `||`.

1.2 Parser

My parser uses Parsec. It assumes left associativity where I have left the associativity ambiguous.

Most of the parser is fairly straightforward. The only interesting pieces of code are those handling precedence and associativity in `command'` and `expr'`. I have used Parsec's `chainl1` to handle `Command '||' Command`, and my own `chainpre1` to handle `Command '@' Vident`, `Expr '+' Prim`, and `Expr '-' Prim`.

`chainl1` is used to run the parsers of highest precedence first and then join them with lower precedence operators. It runs on expressions of the type `a ::= a op a`. `chainpre1` does the same, except it works on expressions of the type `a ::= a op b` instead.

1.3 Tests

To test the correctness of the parser, it is necessary to keep in mind the ambigious grammar. What's important is that the semantics of a parsed program are right, so instead of testing by comparing parsed programs to ASTs, I have tested by comparing interpreted parsed program to animations. The downside is that the tests depend on the correctness of the interpreter, but there is no other practical way unless the grammar is made unambigious.

I have made two sets af manual tests and one QuickCheck property in `SalsaParserTests.hs`.

The first set of manual tests checks the animation interpreted from a parsed string. The second set tests that strings with wrong grammar are not succesfully parsed.

The QuickCheck property generates a program AST, formats it to a string, parses the string into a new AST, interprets both ASTs and compares the resulting animations. This is done 1000 times. Read more about my program generator in section 4.

All tests pass, so my implementation is probably sound. However, I cannot be sure until I have tested my interpreter, since it is used in the parser tests.

2 Interpreting Salsa

In my implementation, a shape is only drawn if a move happens, i.e. an animation with just one key frame has 0 frames. I don't think this was clear from the exam text, but my way seems the simplest and fits nicely with the required functionality of `interpolate`.

2.1 Salsa and SalsaCommand monads

These are my main types:

```
type Shape = Position -> Ident -> GpxInstr

data Env = Env { views :: [(Ident, Integer, Integer)]
               , groups :: [(Ident, [Ident])]
               , shapes :: M.Map Ident (M.Map Ident Shape)
               , activeViews :: [Ident]
               , envFps :: Integer
               }

-- | The positions of the shapes in the views.
type ShapePosMap = M.Map Ident (M.Map Ident Position)

data Context = Context { contextEnv :: Env
                       , contextSPM :: ShapePosMap
                       }

newtype SalsaCommand a = SalsaCommand {
  runSC :: Context -> (ShapePosMap, [Frame], a) }

newtype Salsa a = Salsa { runS :: Context -> (Context, [Frame], a) }
```

`Env`, `ShapePosMap` and `Context` holds what's required of them. `SalsaCommand` and `Salsa` are almost the same (also in their monad instances), the only difference being that `Salsa` can modify all of the `Context`, and that `SalsaCommand` can only modify the `ShapePosMap`. This is because a Salsa `Command` should only move shapes and generate frames, not change the context in any way.

Since `Salsa` and `SalsaCommand` are so alike, they can share several useful functions. For this reason I have made a typeclass `SalsaSomething` with the functions `getContext` and `modifySPM` which is instanced for both types.

2.2 Generating frames

The interesting part of the interpreter is how it generates frames. `runProg` starts a `Salsa` monad which, as long as there are `Definitions`, just modifies the context state again and again. When it sees a `Command`, the `SalsaCommand` monad generates frames which are then appended to the existing frames in the `Salsa` monad state.

For a given `Command`, `SalsaCommand` generates frames this way:

1. Save the old shape positions.

2. Create as many parallel commands as needed so that every command is simplified to a move of just one shape. If there is an `At`, it needs to be run in another environment, which is done by the `withEnv` function.
3. For each shape move, calculate and set the new position for the shape in all its active views.
4. For each view, for each shape in that view, interpolate the frames to the next key frame, and add them to the `SalsaCommand` state.
5. Merge all the frame lists so that all lists will be executed concurrently when run in an animation.

The bind operation of the monad concatenates frames, so to merge them instead I have created another function, `joinFrames`, which is similar to `>>` except for merging.

2.3 Tests

I have created a set of manual tests and a set of QuickCheck properties in `SalsaInterpTests.hs`.

The manual tests each take an fps number, a program string, and the expected `Animation` and checks that the parsed program interpreted at the given fps results in the expected animation. I have made manual tests for interpolating, shapes, multiple moves, views and groups, the `@` and `||` operators, and `Expr`.

The QuickCheck properties test the invariants:

- `interpolate n p0 p1` must return a sorted list where the last element is `p1` if `n /= 0`, and the length is `n`.
- The number of frames must be equal to the number of non-parallel moves times the framerate.
- All views of an AST must also be in the animation.
- When a shape occurs in the AST, it must also occur in at least one frame in the animation (because the program generator makes sure to move all shapes at least once).
- When a move-to-position is absolute, there must be at least one frame in which a shape is at that position.

All tests pass. However, some of the tests depend on the correctness of the parser, and since some of the parser tests depend on the correctness of the interpreter, there is a co-dependency problem here. But since the QuickCheck properties in both the parser and interpreter are checked alongside manual tests which show that their outputs are clearly different for different inputs, it's probably not a problem, so my implementation is probably sound.

All Salsa parser and interpreter tests can be run with `test.hs`. You can also run `animate.hs` to generate animations from files.

3 Atomic Transaction Server

The source files are in `src/at_server`.

I use OTP in my implementation. My transaction server is split up into two FSMs: the master (`at_server_master.erl`) and the helper (`at_server_helper.erl`). In `at_server.erl`, the master is directly used for `start/1`, `stop/1`, `doquery/2`, and `begin_t/1`, while helpers are used for `query_t/3`, `update_t/3` and `commit_t/2`.

3.1 Master and helper functionality

There is only one state name, `ready`. When `at_server:start(State)` is called, `State` is stored in the master FSM, and can only changed with a commit message from a helper.

Even though the master aborts all transactions when accepting a commit from a helper process, another helper process might still have had time to send the master another commit. To ensure that that commit is refused, the master and the helpers maintain `Iter` states.

When a transaction is created, the master's `Iter` is copied to the helper, and when the helper tries to commit, the master only accepts it if their `Iter`s are equal. Once accepted, the master increases its `Iter` with 1, thus refusing any further commits from currently dying processes.

The helper also has only state name, `ready`. Like the master it maintains `State` and `Iter` states, but it also keeps a reference to its master.

3.2 PIDs instead of refs

In my implementation, transactions are referred to by PIDs instead of refs. It seems easier and smarter than maintaining a dictionary in the master and having to query the master every time a helper is to run a command.

One upside of the dictionary approach is that a transaction not created by the current master cannot commit to the current master. With direct command passing, that upside isn't there. I have corrected this by making checks in all helper functions used by the API that check whether the internal master pid equals the master pid in the API function call (like `at_server:update_t(Master, ...)`

On reflection, I realize that while refs created by `make_ref` are unique[1], I don't know about process ids. If a new process takes over the id of a previous process, that might cause problems, but it doesn't appear to be a problem in my tests.

3.3 Limitations in `at_extapi`

The requirement that functions in `at_extapi.erl` must contact the server via `at_server.erl` means that `abort` and `ensureUpdate` are not as efficient as they could be.

To abort a transaction, `abort` has to exploit the fact that a transaction must fail if `at_server:update_t/3` fails with an error. To do this, it sends a function to `update_t` which always fail. A simpler way would be to just kill the transaction with `exit(Helper, kill)`.

[1] `http://www.erlang.org/doc/man/erlang.html`

The way `ensureUpdate` works is that it calls `tryUpdate` again and again until it returns `ok` or `error`. A simpler, less forceful way would be to extend the main API to accept updates directly on the master, although that would remove the concurrency.

It is not specified in the exam text what to return in `choiceUpdate` if all return values are `error`. In my implementation, the final return value is then also `error`.

3.4 Ensuring `aborted`

Assuming that no new process takes the pid of a previous process, a call to an aborted transaction will always return `aborted`, since the implementation catches `noproc` errors. However, a transaction might not **be** aborted at once, since `at_extapi:abort/2` is used to abort the processes.

3.5 Tests

I have added type specifications to most functions with `-spec`, although I haven't tested them with Dialyzer; I put them there mostly for my own sake.

The file `at_server_tests.erl` contains 6 hand-crafted tests of the transaction server which all pass. The tests use the EUnit testing framework[2] and can be run with the generated function `at_server_tests:test/0`.

They test basic function application, sequential transactions, concurrent transactions, `tryUpdate`, `ensureUpdate`, and `choiceUpdate`. That covers all of the API, so it seems my implementation is correct.

[2] `http://www.erlang.org/doc/apps/eunit/chapter.html`

4 Salsa program generator

See `SalsaGenerator.hs`.

For some of the Salsa tests to work, there must be a program generator to create random Salsa ASTs. For some tests there must also be a program text generator to transform an AST into a string.

To create the random ASTs, I use QuickCheck's `Test.QuickCheck.Gen` module. The generation is fairly straightforward, except that the generated programs need to be correct for some of the tests to make sense. To ensure correctness, the generator keeps a environment like in the interpreter (but simpler), so that it always knows which shapes it can move, etc.

Creating a program string from an AST is straightforward, though it's important to remember that the left associativity in `Expr '-' Prim` might make it necessary to put parantheses around an expression. There are infinitely many correct program texts matching a single AST because expressions can be in arbitrarily many parentheses, and because there can be arbitrarily many whitespace characters. My random program string generator uses this to create strings with few or many spaces, parentheses and brackets, so that the parser can be fed differently formatted versions of the same program.

A Source Code

A.1 Salsa

A.1.1 General

Gpx.hs

```
{-# OPTIONS_GHC -Wall -fno-warn-unused-do-bind #-}
{-------------------------------------------------------------------
Gpx.  Haven't changed this.
-------------------------------------------------------------------
Student name: Niels G. W. Serup <ngws@metanohi.name>
Student KU-id: njf941
-------------------------------------------------------------------}
module Gpx where

type ViewName = String
type ColourName = String
type Frame = [GpxInstr]
type Animation = ([(ViewName, Integer, Integer)], [Frame])
data GpxInstr = DrawRect Integer Integer Integer Integer ViewName ColourName
              | DrawCirc Integer Integer Integer ViewName ColourName
              deriving (Eq, Show, Ord)
```

Misc.hs

```
{-# OPTIONS_GHC -Wall -fno-warn-unused-do-bind #-}
{-# LANGUAGE FlexibleContexts #-}
{-------------------------------------------------------------------
Miscellaneous values and functions useful in the Salsa universe.
-------------------------------------------------------------------
Student name: Niels G. W. Serup <ngws@metanohi.name>
Student KU-id: njf941
-------------------------------------------------------------------}
module Misc where

import Text.Parsec.Prim
import Text.Parsec.Char
import Text.Parsec.Combinator
import Control.Applicative hiding (Const, (<|>), many)
import Control.Monad
import Test.QuickCheck.Gen
import System.Random (StdGen)
import Data.Maybe (fromMaybe)

import SalsaAst
import Gpx


-------------------------------------------------------------------
--- Misc. core
-------------------------------------------------------------------

type Position = (Integer, Integer)
```

```
reservedWords :: [String]
reservedWords = ["viewdef", "rectangle", "circle", "group", "view"]
                ++ map fst colours

colours :: [(String, Colour)]
colours = [ ("blue", Blue)
          , ("plum", Plum)
          , ("red", Red)
          , ("green", Green)
          , ("orange", Orange)
          ]


--------------------------------------------------------------------------
--- Misc. extensions
--------------------------------------------------------------------------

(<:>) :: Applicative f => f a -> f [a] -> f [a]
x <:> xs = (:) <$> x <*> xs

(<++>) :: Applicative f => f [a] -> f [a] -> f [a]
xs <++> ys = (++) <$> xs <*> ys

trail :: Int -> [[a]] -> [[a]]
trail n xs = xs ++ replicate (max 0 (n - length xs)) []

getColourName :: Colour -> String
getColourName col = fromMaybe (error "colour does not exist")
                    $ lookup col $ map (\(a, b) -> (b, a)) colours

getColour :: String -> Colour
getColour col = fromMaybe (error "colour does not exist")
                $ lookup col colours

nFrames :: Animation -> Int
nFrames (_, frms) = length frms

tryEvalExpr :: Expr -> Maybe Integer
tryEvalExpr e = case e of
  Plus a b -> (+) <$> tryEvalExpr a <*> tryEvalExpr b
  Minus a b -> (-) <$> tryEvalExpr a <*> tryEvalExpr b
  Const a -> return a
  Xproj _ -> Nothing
  Yproj _ -> Nothing


--------------------------------------------------------------------------
--- QuickCheck extensions
--------------------------------------------------------------------------

-- | Generates a value.
genVal :: StdGen -> Gen a -> a
genVal r g = unGen g r 0


--------------------------------------------------------------------------
--- Parsec extensions
```

```
-----------------------------------------------------------------------

whiteSpace :: Stream s m Char => ParsecT s u m ()
whiteSpace = void (space <|> char '\n')

whiteSpaces :: Stream s m Char => ParsecT s u m ()
whiteSpaces = void (many whiteSpace)

lexeme :: Stream s m Char => ParsecT s u m a -> ParsecT s u m a
lexeme p = p <* whiteSpaces

symbol :: Stream s m Char => String -> ParsecT s u m String
symbol = lexeme . try . string

inParens :: Stream s m Char => ParsecT s u m a -> ParsecT s u m a
inParens = lexeme . between (lexeme $ char '(') (char ')')

inBrackets :: Stream s m Char => ParsecT s u m a -> ParsecT s u m a
inBrackets = lexeme . between (lexeme $ char '{') (char '}')

inBraces :: Stream s m Char => ParsecT s u m a -> ParsecT s u m a
inBraces = lexeme . between (lexeme $ char '[') (char ']')

-- | Parse a chain of infix expressions (a ::= a op b).
chainpre1 :: Stream s m Char =>
             ParsecT s u m a ->
             ParsecT s u m (a -> a) ->
             ParsecT s u m a
chainpre1 p0 op = p0 >>= chain'
  where chain' t = (try (($ t) <$> op) >>= chain')
                   <|> return t
```

SalsaAst.hs

```
{-# OPTIONS_GHC -Wall -fno-warn-unused-do-bind #-}
{-----------------------------------------------------------------------
SalsaAst.  Haven't changed this except for SIdent and VIdent additions.
-----------------------------------------------------------------------
Student name: Niels G. W. Serup <ngws@metanohi.name>
Student KU-id: njf941
-----------------------------------------------------------------------}
module SalsaAst where

type Program = [DefCom]
data DefCom = Def Definition
            | Com Command
            deriving (Show, Eq)
data Definition = Viewdef VIdent Expr Expr
                | Rectangle SIdent Expr Expr Expr Expr Colour
                | Circle SIdent Expr Expr Expr Colour
                | View VIdent
                | Group VIdent [VIdent]
                deriving (Show, Eq)
data Command = Move [SIdent] Pos
             | At Command Ident
             | Par Command Command
             deriving (Show, Eq)
```

```
data Pos = Abs Expr Expr
         | Rel Expr Expr
         deriving (Show, Eq)
data Expr = Plus Expr Expr
          | Minus Expr Expr
          | Const Integer
          | Xproj SIdent
          | Yproj SIdent
          deriving (Show, Eq)
data Colour = Blue | Plum | Red | Green | Orange
            deriving (Show, Eq)
type Ident = String

type SIdent = Ident
type VIdent = Ident
```

Salsa.hs

```
{-# OPTIONS_GHC -Wall -fno-warn-unused-do-bind #-}
{-------------------------------------------------------------------
General Salsa API.
-------------------------------------------------------------------
Student name: Niels G. W. Serup <ngws@metanohi.name>
Student KU-id: njf941
-------------------------------------------------------------------}
module Salsa (
  parseString,
  parseFile,
  runProg,
  runString,
  runFile
  ) where

import Control.Applicative ((<$>))

import SalsaParser (parseString, parseFile)
import SalsaInterp (runProg)
import Gpx

runString :: Integer -> String -> Animation
runString fps s = runProg fps $ either (error . show) id $ parseString s

runFile :: Integer -> FilePath -> IO Animation
runFile fps p = runProg fps . either (error . show) id <$> parseFile p
```

A.1.2 Parser

SalsaParser.hs

```
{-# OPTIONS_GHC -Wall -fno-warn-unused-do-bind #-}
{-------------------------------------------------------------------
Salsa parser.
-------------------------------------------------------------------
Student name: Niels G. W. Serup <ngws@metanohi.name>
Student KU-id: njf941
-------------------------------------------------------------------}
```

```
module SalsaParser (
  parseString,
  parseFile,
  Error
  ) where


import Text.Parsec.Prim
import Text.Parsec.Char
import Text.Parsec.Error
import Text.Parsec.String
import Text.Parsec.Combinator
import Control.Monad
import Control.Applicative (Applicative, (<$>), (<*>), (<*))
import Data.Char

import Misc
import SalsaAst


type Error = ParseError


program :: Parser Program
program = whiteSpaces >> lexeme (many1 defCom) <* eof
          <?> "Program"


defCom :: Parser DefCom
defCom = lexeme defCom' <?> "DefCom"

defCom' :: Parser DefCom
defCom' = Def <$> definition
          <|> Com <$> command


definition :: Parser Definition
definition = lexeme definition' <?> "Definition"

definition' :: Parser Definition
definition' = choice [viewdef, view, rectangle, circle, group]
  where viewdef = lexeme (try (string "viewdef") >> whiteSpace)
                  >> Viewdef <$> vIdent <*> expr <*> expr
        rectangle = lexeme (try (string "rectangle") >> whiteSpace)
                    >> Rectangle <$> sIdent
                    <*> expr <*> expr <*> expr <*> expr <*> colour
        circle = lexeme (try (string "circle") >> whiteSpace)
                 >> Circle <$> sIdent
                 <*> expr <*> expr <*> expr <*> colour
        view = lexeme (try (string "view") >> whiteSpace)
               >> View <$> vIdent
        group = lexeme (try (string "group") >> whiteSpace)
                >> Group <$> vIdent <*> inBraces (many1 vIdent)


command :: Parser Command
command = lexeme command' <?> "Command"
```

```
command' :: Parser Command
command' = par
  where par = chainl1 at (symbol "||" >> return Par)
        at = chainpre1 (move <|> inBrackets command)
             (symbol "@" >> fmap (flip At) vIdent)
        move = do
          is <- many1 sIdent
          symbol "->"
          p <- pos
          return $ Move is p


vIdent :: Parser Ident
vIdent = lexeme vIdent' <?> "VIdent"

vIdent' :: Parser Ident
vIdent' = satisfy isUpper <:> baseIdent


sIdent :: Parser Ident
sIdent = lexeme (try sIdent') <?> "SIdent"

sIdent' :: Parser Ident
sIdent' = do
  i <- satisfy isLower <:> baseIdent
  guard (i `notElem` reservedWords)
  return i


baseIdent :: Parser Ident
baseIdent = many (alphaNum <|> char '_')


pos :: Parser Pos
pos = lexeme pos' <?> "Pos"

pos' :: Parser Pos
pos' = absp <|> relp
  where absp = inParens $ Abs <$> expr <*> (symbol "," >> expr)
        relp = symbol "+" >> fmap (\(Abs a b) -> Rel a b) absp


expr :: Parser Expr
expr = lexeme expr' <?> "Expr"

expr' :: Parser Expr
expr' = chainpre1 prim $ choice
        [ symbol "+" >> fmap (flip Plus) prim
        , symbol "-" >> fmap (flip Minus) prim
        ]

prim :: Parser Expr
prim = lexeme prim' <?> "Prim"

prim' :: Parser Expr
prim' = integer <|> dot <|> inParens expr
```

```
  where integer = Const . read <$> many1 digit
        dot = do
          i <- sIdent <* lexeme (char '.')
          choice [ char 'x' >> return (Xproj i)
                 , char 'y' >> return (Yproj i)
                 ]


colour :: Parser Colour
colour = lexeme colour' <?> "Colour"

colour' :: Parser Colour
colour' = choice $ map toRule colours
  where toRule (name, constr) =
          lexeme (string name >> (void whiteSpace <|> eof)) >> return constr


-- | Parse a Salsa program.
parseString :: String -> Either Error Program
parseString = parse program "<stdin>"

-- | Parse a Salsa program from file.
parseFile :: FilePath -> IO (Either Error Program)
parseFile = parseFromFile program
```

A.1.3 Interpreter

SalsaInterp.hs

```
{-# OPTIONS_GHC -Wall -fno-warn-unused-do-bind #-}
{-# LANGUAGE TupleSections #-}
{-------------------------------------------------------------------
Salsa interpreter.
-------------------------------------------------------------------
Student name: Niels G. W. Serup <ngws@metanohi.name>
Student KU-id: njf941
-------------------------------------------------------------------}
module SalsaInterp where

import qualified Data.Map as M
import Data.Maybe (mapMaybe, fromMaybe)
import Control.Applicative hiding (Const)
import Control.Monad
import Control.Arrow (second)

import Misc
import SalsaAst
import Gpx


-------------------------------------------------------------------
--- Types
-------------------------------------------------------------------

type Shape = Position -> VIdent -> GpxInstr
```

```
data Env = Env { views :: [(VIdent, Integer, Integer)]
               , groups :: [(VIdent, [VIdent])]
               , shapes :: M.Map VIdent (M.Map SIdent Shape)
               , activeViews :: [VIdent]
               , envFps :: Integer
               }

-- | The positions of the shapes in the views.
type ShapePosMap = M.Map VIdent (M.Map SIdent Position)

data Context = Context { contextEnv :: Env
                       , contextSPM :: ShapePosMap
                       }

newtype SalsaCommand a = SalsaCommand {
  runSC :: Context -> (ShapePosMap, [Frame], a) }

newtype Salsa a = Salsa { runS :: Context -> (Context, [Frame], a) }


--------------------------------------------------------------------------
--- Class glue for common functionality
--------------------------------------------------------------------------

class SalsaSomething s where
  getContext :: s Context
  modifySPM :: (ShapePosMap -> ShapePosMap) -> s ()


--------------------------------------------------------------------------
--- Instances of Monad, Functor and Applicative
--------------------------------------------------------------------------

instance Monad SalsaCommand where
  return a = SalsaCommand $ \(Context _ sm) -> (sm, [], a)

  x >>= g = SalsaCommand f
    where f ctx0@(Context env _) = (sm2, frms2, b)
            where (sm1, frms0, a) = runSC x ctx0
                  ctx1 = Context env sm1
                  (sm2, frms1, b) = runSC (g a) ctx1
                  frms2 = frms0 ++ frms1

instance Functor SalsaCommand where
  fmap = liftM

instance Applicative SalsaCommand where
  pure = return
  (<*>) = ap

instance SalsaSomething SalsaCommand where
  getContext = SalsaCommand $ \ctx@(Context _ sm) -> (sm, [], ctx)
  modifySPM sf = SalsaCommand $ \(Context _ sm) -> (sf sm, [], ())


instance Monad Salsa where
  return a = Salsa $ \ctx -> (ctx, [], a)
```

```
  x >>= g = Salsa f
    where f ctx0 = (ctx2, frms2, b)
            where (ctx1, frms0, a) = runS x ctx0
                  (ctx2, frms1, b) = runS (g a) ctx1
                  frms2 = frms0 ++ frms1

instance Functor Salsa where
  fmap = liftM

instance Applicative Salsa where
  pure = return
  (<*>) = ap

instance SalsaSomething Salsa where
  getContext = Salsa $ \ctx -> (ctx, [], ctx)
  modifySPM sf = modifyContext (\(Context env sm) -> Context env (sf sm))


-----------------------------------------------------------------------
--- General functions
-----------------------------------------------------------------------

-- | Interpolate positions.  Exclude the start position.
interpolate :: Integer -> Position -> Position -> [Position]
interpolate 0 _ _ = []
interpolate n p0 p1 | p0 < p1 = interpolate' p0 p1
                    | otherwise = tail (reverse $ interpolate' p1 p0) ++ [p1]
  where interpolate' (startX, startY) (endX, endY)
          = zip (makeList startX endX) (makeList startY endY)
        makeList start end = map (round . (+ fromIntegral start)
                                  . (* step) . fromIntegral) [1..n - 1]
                             ++ [end]
          where step :: Double
                step = fromIntegral (end - start) / fromIntegral n

emptyEnv :: Env
emptyEnv = Env { views = []
               , groups = []
               , shapes = M.empty
               , activeViews = []
               , envFps = 1
               }

emptySPM :: ShapePosMap
emptySPM = M.empty


-----------------------------------------------------------------------
--- SalsaSomething functions
-----------------------------------------------------------------------

-- | Get the shape positions in the active views.
getShapePos :: (SalsaSomething s, Monad s) => SIdent -> s [(VIdent, Position)]
getShapePos i = do
  Context env sm <- getContext
  return $ mapMaybe (look sm) $ activeViews env
```

```
  where look sm vi = (vi,) <$> (M.lookup vi sm >>= M.lookup i)

-- | Get the shape positions in all views.
getAllShapePos :: (SalsaSomething s, Monad s) => SIdent -> s [(VIdent, Position)]
getAllShapePos i = do
  Context env sm <- getContext
  return $ mapMaybe (look sm . \(a, _, _) -> a) $ views env
  where look sm vi = (vi,) <$> (M.lookup vi sm >>= M.lookup i)

evalExpr :: (SalsaSomething s, Functor s, Applicative s, Monad s)
            => Expr -> s Integer
evalExpr expr = case expr of
  Plus e0 e1 -> (+) <$> evalExpr e0 <*> evalExpr e1
  Minus e0 e1 -> (-) <$> evalExpr e0 <*> evalExpr e1
  Const n -> return n
  Xproj si -> fst <$> lowestCoordinates si
  Yproj si -> snd <$> lowestCoordinates si
  where lowestCoordinates si = do
          ts <- map snd <$> getAllShapePos si
          case ts of
            [] -> error ("no shape '" ++ si ++ "' defined yet")
            _ -> return $ minimum ts

getEnv :: (SalsaSomething s, Functor s) => s Env
getEnv = contextEnv <$> getContext

getSPM :: (SalsaSomething s, Functor s) => s ShapePosMap
getSPM = contextSPM <$> getContext



------------------------------------------------------------------------
--- SalsaCommand functions
------------------------------------------------------------------------

-- | Join the frames generated by one command with the frames generated by
-- another.  Use the ShapePosMap of the latter.
joinFrames :: SalsaCommand () -> SalsaCommand () -> SalsaCommand ()
joinFrames x y = SalsaCommand f
  where f ctx = (sm1, frms2, ())
          where (_, frms0, ()) = runSC x ctx
                (sm1, frms1, ()) = runSC y ctx
                frms2 = zipWith (++) (trail (length frms1) frms0)
                        (trail (length frms0) frms1)

-- | Run a command in another environment.
withEnv :: (Env -> Env) -> SalsaCommand a -> SalsaCommand a
withEnv f c = SalsaCommand $ \(Context env sm) -> runSC c $ Context (f env) sm

addFrame :: Frame -> SalsaCommand ()
addFrame frm = SalsaCommand $ \(Context _ sm) -> (sm, [frm], ())

putShapePos :: SalsaSomething s => VIdent -> SIdent -> Position -> s ()
putShapePos vi si absP = modifySPM g
  where g spm = M.update (Just . M.insert si absP) vi spm'
          where spm' | vi `M.notMember` spm = M.insert vi M.empty spm
                     | otherwise = spm
```

```
updateShapePos :: SIdent -> (Position -> SalsaCommand Position) -> SalsaCommand ()
updateShapePos si pf = do
  ss <- getShapePos si
  forM_ ss $ \(vi, p0) -> do
    p1 <- pf p0
    putShapePos vi si p1

makeShapeGpx :: VIdent -> SIdent -> Position -> SalsaCommand GpxInstr
makeShapeGpx vi si p = do
  ss <- shapes <$> getEnv
  let s = fromMaybe
          (fail ("no shape of the name '" ++ si ++ "' in view '" ++ vi ++ "'"))
          (M.lookup si =<< M.lookup vi ss)
  return $ s p vi

-- | Add all frames.
commitFrames :: ShapePosMap -> SalsaCommand ()
commitFrames old = foldl joinFrames (return ())
                   $ map (createViewFrames . second M.toList)
                   $ M.toList old

-- | Create all frames for a view, using the old positions of the shapes in the
-- view.
createViewFrames :: (VIdent, [(SIdent, Position)]) -> SalsaCommand ()
createViewFrames (vi, ss) = foldl joinFrames (return ())
                            $ map (createShapeFrames vi) ss

-- | Create all frames for a shape in a view, using the old position of the
-- shape.
createShapeFrames :: VIdent -> (SIdent, Position) -> SalsaCommand ()
createShapeFrames vi (si, p0) = do
  spm <- getSPM
  fps <- envFps <$> getEnv
  let p1 = fromMaybe
           (error ("shape '"
                   ++ si ++ "' in view '"
                   ++ vi ++ "' not present in the present"))
           (M.lookup si =<< M.lookup vi spm)
      ps = interpolate fps p0 p1
  forM_ ps (addFrame . (: []) <=< makeShapeGpx vi si)

move :: SIdent -> Pos -> SalsaCommand ()
move i p = updateShapePos i $ \(x0, y0) -> case p of
  Abs x y -> (,) <$> evalExpr x <*> evalExpr y
  Rel xd yd -> do
    xd' <- evalExpr xd
    yd' <- evalExpr yd
    return (x0 + xd', y0 + yd')

command' :: Command -> SalsaCommand ()
command' c = case c of
  Move is p -> mapM_ (`move` p) is
  At c1 i -> withEnv (\env -> env { activeViews = [i] }) $ command' c1
  Par c0 c1 -> command' c0 >> command' c1

command :: Command -> SalsaCommand ()
```

```
command c = do
  old <- getSPM
  command' c
  commitFrames old


-----------------------------------------------------------------------
--- Salsa commands
-----------------------------------------------------------------------

putContext :: Context -> Salsa ()
putContext ctx = Salsa $ const (ctx, [], ())

modifyContext :: (Context -> Context) -> Salsa ()
modifyContext cf = putContext =<< cf <$> getContext

modifyEnv :: (Env -> Env) -> Salsa ()
modifyEnv ef = modifyContext (\(Context env sm) -> Context (ef env) sm)

addView :: VIdent -> Integer -> Integer -> Salsa ()
addView vi w h = modifyEnv $ \env -> env { views = (vi, w, h) : views env }

addGroup :: VIdent -> [VIdent] -> Salsa ()
addGroup gi vis = modifyEnv $ \env -> env { groups = (gi, vis) : groups env }

setActiveViews :: VIdent -> Salsa ()
setActiveViews i = do
  av <- fromMaybe [i] . lookup i . groups <$> getEnv
  modifyEnv $ \env -> env { activeViews = av }

-- | Add a shape the Env and its position to the ShapePosMap.
addShape :: SIdent -> Shape -> Salsa ()
addShape si s = do
  av <- activeViews <$> getEnv
  forM_ av $ \vi ->
    modifyEnv $ \env ->
      if M.notMember vi $ shapes env
      then env { shapes = M.insert vi M.empty $ shapes env }
      else env

  forM_ av $ \vi -> modifyEnv $ \env -> env {
    shapes = M.update (Just . M.insert si s) vi $ shapes env
    }

liftC :: SalsaCommand a -> Salsa a
liftC c = Salsa f
  where f ctx@(Context env _) = (Context env sm, frms, a)
          where (sm, frms, a) = runSC c ctx

definition :: Definition -> Salsa ()
definition d = case d of
  Viewdef vi w h -> do
    w' <- evalExpr w
    h' <- evalExpr h
    addView vi w' h'
    setActiveViews vi
  View i -> setActiveViews i
```

```
  Group gi vis -> addGroup gi vis
  Rectangle si x y w h col -> do
    x' <- evalExpr x
    y' <- evalExpr y
    w' <- evalExpr w
    h' <- evalExpr h
    addShape si $ \(x1, y1) vi -> DrawRect x1 y1 w' h' vi $ getColourName col
    vis <- activeViews <$> getEnv
    forM_ vis $ \vi -> putShapePos vi si (x', y')
  Circle si x y r col -> do
    x' <- evalExpr x
    y' <- evalExpr y
    r' <- evalExpr r
    addShape si $ \(x1, y1) vi -> DrawCirc x1 y1 r' vi $ getColourName col
    vis <- activeViews <$> getEnv
    forM_ vis $ \vi -> putShapePos vi si (x', y')

defCom :: DefCom -> Salsa ()
defCom (Def d) = definition d
defCom (Com c) = liftC $ command c


-- | Generate an animation from a program.
runProg :: Integer -> Program -> Animation
runProg fps prog = (views $ contextEnv ctx1, frms)
  where ctx = Context (emptyEnv { envFps = fps }) emptySPM
        (ctx1, frms, ()) = runS makeFrames ctx
        makeFrames = mapM_ defCom prog
```

A.1.4 Tests

SalsaGenerator.hs

```
{-# OPTIONS_GHC -Wall -fno-warn-unused-do-bind #-}
{-------------------------------------------------------------------
Correct Salsa Program generator.
-------------------------------------------------------------------
Student name: Niels G. W. Serup <ngws@metanohi.name>
Student KU-id: njf941
-------------------------------------------------------------------}
module SalsaGenerator where

import Control.Applicative hiding (Const)
import Test.QuickCheck
import Control.Monad.State
import qualified Data.Map as M
import Data.Maybe (fromMaybe)
import Data.List (delete, nub)

import Misc
import SalsaAst


-------------------------------------------------------------------
--- General functions for generating programs
-------------------------------------------------------------------
```

```
data MiniEnv = MiniEnv { depth :: Int
                       , views :: [(Ident, [Ident])]
                       , shapes :: M.Map Ident [Ident]
                       , notMovedShapes :: [(Ident, Ident)]
                       , activeView :: Ident
                       , noProj :: Bool -- disable Xproj and Yproj generation
                       }

emptyMiniEnv :: MiniEnv
emptyMiniEnv = MiniEnv { depth = 0
                       , views = []
                       , shapes = M.empty
                       , notMovedShapes = []
                       , activeView = error "not defined"
                       , noProj = False
                       }

type GenBuilder = StateT MiniEnv Gen

incDepth :: GenBuilder ()
incDepth = modify $ \env -> env { depth = depth env + 1 }

oneof' :: [GenBuilder a] -> GenBuilder a
oneof' = join . lift . elements

pickSomeBetween :: Eq a => Int -> Int -> [a] -> Gen [a]
pickSomeBetween a b xs = do
  n <- choose (a, b)
  nub <$> replicateM n (elements xs)

addView :: Ident -> [Ident] -> GenBuilder ()
addView gi vis = modify $ \env -> env { views = (gi, vis) : views env }

setActiveView :: Ident -> GenBuilder ()
setActiveView vi = modify $ \env -> env { activeView = vi }

getActiveViews :: GenBuilder [Ident]
getActiveViews = do
  env <- get
  return $ fromMaybe (error "meh") $ lookup (activeView env) (views env)

getAllViews :: GenBuilder [Ident]
getAllViews = concatMap snd . views <$> get

getAllActiveShapes :: GenBuilder [Ident]
getAllActiveShapes = do
  vis <- getActiveViews
  sis <- shapes <$> get
  return $ concatMap (\vi -> fromMaybe [] $ M.lookup vi sis) vis

addShape :: Ident -> GenBuilder ()
addShape si = do
  vis <- getActiveViews
  forM_ vis $ \vi ->
    modify $ \env -> env { shapes = M.update upd vi $ prep vi $ shapes env
                         , notMovedShapes = (vi, si) : notMovedShapes env
```

```
                        }
    where upd ss = Just (si : ss)
          prep vi ss | M.member vi ss = ss
                     | otherwise = M.insert vi [] ss

fork :: GenBuilder a -> GenBuilder a
fork builder = do
  old <- get
  res <- builder
  put old
  return res

workDepth :: GenBuilder a -> GenBuilder a -> GenBuilder a
workDepth fallback builder = do
  d <- depth <$> get
  if d >= depthMax
    then fallback
    else incDepth >> builder


--------------------------------------------------------------------------
--- Correct arbitrary program generation
--------------------------------------------------------------------------

arbitraryProgram :: Gen Program
arbitraryProgram = evalStateT arbitraryProgramB emptyMiniEnv

arbitraryProgramNoProj :: Gen Program
arbitraryProgramNoProj = evalStateT arbitraryProgramB
                         $ emptyMiniEnv { noProj = True }

depthMax :: Int
depthMax = 5

arbitraryProgramB :: GenBuilder Program
arbitraryProgramB = do
  nDefComs <- lift $ choose (1, 22)
  replicateM nDefComs arbitraryDefComB <++> addEventualMoves

addEventualMoves :: GenBuilder Program
addEventualMoves = do
  ss <- notMovedShapes <$> get
  xs <- forM ss $ \(vi, si) -> do
    p <- arbitraryPosB
    return [ Def $ View vi
           , Com $ Move [si] p
           ]
  return $ concat xs

arbitraryDefComB :: GenBuilder DefCom
arbitraryDefComB = do
  vs <- views <$> get
  ss <- getAllActiveShapes
  if null vs
    then Def <$> arbitraryDefinitionB
    else oneof' [ Def <$> arbitraryDefinitionB
                , if null ss
```

```
                then Def <$> arbitraryDefinitionB
                else Com <$> arbitraryCommandB
              ]

arbitraryDefinitionB :: GenBuilder Definition
arbitraryDefinitionB = do
  vs <- views <$> get
  if null vs
    then viewdef
    else do
    ss <- getAllActiveShapes
    oneof' $ if null ss
             then [rectangle, circle]
             else [viewdef, rectangle, circle, view, group]
  where viewdef = do
          i <- arbitraryVIdentB
          addView i [i]
          setActiveView i
          Viewdef i <$> arbitraryExprB <*> arbitraryExprB

        rectangle = do
          i <- arbitrarySIdentB
          r <- Rectangle i <$> arbitraryExprB <*> arbitraryExprB
               <*> arbitraryExprB <*> arbitraryExprB
               <*> arbitraryColourB
          addShape i
          return r

        circle = do
          i <- arbitrarySIdentB
          c <- Circle i <$> arbitraryExprB <*> arbitraryExprB
               <*> arbitraryExprB <*> arbitraryColourB
          addShape i
          return c

        view = do
          vis <- map fst . views <$> get
          vi <- lift $ elements vis
          setActiveView vi
          return $ View vi

        group = do
          gi <- arbitraryVIdentB
          vis <- map fst . views <$> get
          vis' <- lift $ pickSomeBetween 1 4 vis
          return $ Group gi vis'

arbitraryCommandB :: GenBuilder Command
arbitraryCommandB = oneof' [move, at, par]
  where move = do
          sis <- getAllActiveShapes
          sis' <- lift $ pickSomeBetween 1 4 sis
          av <- getActiveViews
          forM_ av $ \vi -> forM_ sis' $ \si ->
            modify $ \env -> env { notMovedShapes = delete (vi, si)
                                                    $ notMovedShapes env }
          Move sis' <$> arbitraryPosB
```

```
        at = do
          vis <- getAllViews
          vi <- lift $ elements vis
          workDepth (At <$> move <*> return vi)
            (At <$> fork arbitraryCommandB <*> return vi)

        par = workDepth (Par <$> move <*> move)
              (Par <$> fork arbitraryCommandB <*> fork arbitraryCommandB)

arbitraryPosB :: GenBuilder Pos
arbitraryPosB = oneof' [absPos, relPos]
  where absPos = Abs <$> fork arbitraryExprB <*> fork arbitraryExprB
        relPos = Rel <$> fork arbitraryExprB <*> fork arbitraryExprB

arbitraryVIdentB :: GenBuilder Ident
arbitraryVIdentB = do
  vis <- map fst . views <$> get
  len <- lift $ choose (1, 17)
  lift $ suchThat (elements ['A'..'Z'] <:> replicateM len arbitraryChar)
    (`notElem` vis)

arbitrarySIdentB :: GenBuilder Ident
arbitrarySIdentB = do
  vis <- map fst . views <$> get
  len <- lift $ choose (1, 17)
  lift $ suchThat (elements ['a'..'z'] <:> replicateM len arbitraryChar)
    (`notElem` (vis ++ reservedWords))

arbitraryValidSIdentB :: GenBuilder Ident
arbitraryValidSIdentB = lift . elements =<< getAllActiveShapes

arbitraryChar :: Gen Char
arbitraryChar = elements ("_" ++ ['a'..'z'] ++ ['A'..'Z'] ++ ['0'..'9'])

arbitraryColour :: Gen Colour
arbitraryColour = elements $ map snd colours

arbitraryColourB :: GenBuilder Colour
arbitraryColourB = lift arbitraryColour

arbitraryExprB :: GenBuilder Expr
arbitraryExprB = do
  no <- noProj <$> get
  oneof' $ if no
            then [plus, minus, constb]
            else [plus, minus, constb, xproj, yproj]
  where plus = workDepth (Plus <$> stopper <*> stopper)
               (Plus <$> fork arbitraryExprB <*> fork arbitraryExprB)
        minus = workDepth (Minus <$> stopper <*> stopper)
                (Minus <$> fork arbitraryExprB <*> fork arbitraryExprB)
        constb = Const <$> arbitraryNumberB
        xproj = proj Xproj
        yproj = proj Yproj
        proj c = do
          ss <- getAllActiveShapes
          if null ss
```

```
            then constb
            else c <$> lift (elements ss)
        stopper = do
          no <- noProj <$> get
          if no
            then constb
            else oneof' [constb, xproj, yproj]


arbitraryNumber :: Gen Integer
arbitraryNumber = choose (0, 10^(7 :: Integer) :: Integer)

arbitraryNumberB :: GenBuilder Integer
arbitraryNumberB = lift arbitraryNumber

arbitraryPosition :: Gen Position
arbitraryPosition = (,) <$> arbitraryNumber <*> arbitraryNumber


--------------------------------------------------------------------------
--- Formatting
--------------------------------------------------------------------------

spaceSep :: [String] -> Gen String
spaceSep ss = foldl (<++>) (return "") $ zipWith spaceSep' ss' $ tail ss'
  where ss' = "" : ss ++ [""]
        spaceSep' s0 s1
          | start0 s0 || start0 s1 = return s0 <++> spaces 0
          | otherwise = return s0 <++> spaces 1
          where start0 s | null s = False
                         | otherwise = last s `elem` "(){}[]@|,."

spaces :: Int -> Gen String
spaces start = do
  n <- choose (start, 1)
  replicateM n (elements " \t\n")

maybeBetween :: String -> String -> Gen String -> Gen String
maybeBetween a b g = oneof [g, f]
  where f = do
          g' <- g
          spaceSep [a, g', b]

showProgram :: Program -> Gen String
showProgram p = spaceSep =<< mapM showDefCom p

showDefCom :: DefCom -> Gen String
showDefCom dc = case dc of
  Def d -> showDefinition d
  Com c -> showCommand c

showDefinition :: Definition -> Gen String
showDefinition d = case d of
  Viewdef i e0 e1 -> do
    e0' <- showExpr e0
    e1' <- showExpr e1
    spaceSep ["viewdef", i, e0', e1']
```

```
  Rectangle i e0 e1 e2 e3 col -> do
    [e0', e1', e2', e3'] <- mapM showExpr [e0, e1, e2, e3]
    spaceSep ["rectangle", i, e0', e1', e2', e3', showColour col]
  Circle i e0 e1 e2 col -> do
    [e0', e1', e2'] <- mapM showExpr [e0, e1, e2]
    spaceSep ["circle", i, e0', e1', e2', showColour col]
  View i -> spaceSep ["view", i]
  Group gi vis -> spaceSep (["group", gi, "["] ++ vis ++ ["]"])

showCommand :: Command -> Gen String
showCommand c = case c of
  Move sis p -> do
    p' <- showPos p
    spaceSep (sis ++ ["->", p'])
  At c1 i -> do
    c1' <- showCommand c1
    c1'' <- case c1 of
      Par _ _ -> spaceSep ["{", c1', "}"]
      _ -> return c1'
    spaceSep [c1'', "@", i]
  Par c0 c1 -> do
    c0' <- showCommand c0
    c1' <- showCommand c1
    spaceSep [c0', "||", c1']

showPos :: Pos -> Gen String
showPos p = case p of
  Abs e0 e1 -> do
    e0' <- showExpr e0
    e1' <- showExpr e1
    spaceSep ["(", e0', ",", e1', ")"]
  Rel e0 e1 -> do
    e0' <- showExpr e0
    e1' <- showExpr e1
    spaceSep ["+", "(", e0', ",", e1', ")"]

showExpr :: Expr -> Gen String
showExpr e = maybeBetween "(" ")" $ case e of
  Plus e0 e1 -> do
    e0' <- showExpr e0
    e1' <- showExpr e1
    spaceSep [e0', "+", e1']
  Minus e0 e1 -> do
    e0' <- showExpr e0
    e1' <- showExpr e1
    let e1'' = case e1 of
          Plus _ _ -> ["(", e1', ")"]
          Minus _ _ -> ["(", e1', ")"]
          _ -> [e1']
    spaceSep ([e0', "-"] ++ e1'')
  Const n -> spaceSep [show n]
  Xproj i -> spaceSep [i, ".", "x"]
  Yproj i -> spaceSep [i, ".", "y"]


showColour :: Colour -> String
showColour = getColourName
```

SalsaParserTests.hs

```haskell
{-# OPTIONS_GHC -Wall -fno-warn-unused-do-bind #-}
{-----------------------------------------------------------------------
Salsa parser tests.
-----------------------------------------------------------------------
Student name: Niels G. W. Serup <ngws@metanohi.name>
Student KU-id: njf941
-----------------------------------------------------------------------}
module SalsaParserTests where

import Test.QuickCheck
import Control.Monad.State
import Data.List (intercalate)

import SalsaAst
import SalsaParser (parseString)
import SalsaInterp (runProg)
import SalsaGenerator


-----------------------------------------------------------------------
--- QuickCheck testing
-----------------------------------------------------------------------

propSemanticallyEqual :: Property
propSemanticallyEqual =
  forAll arbitraryProgram $
  \p -> forAll (showProgram p) $
        \s ->
        case parseString s of
          Right p' -> runProg 1 p == runProg 1 p'
          Left _ -> False


-----------------------------------------------------------------------
--- Other testing
-----------------------------------------------------------------------

rightCheck :: (String, Program) -> IO ()
rightCheck (s, p) = case parseString s of
  Left e -> error ("parse error in:\n" ++ s ++ "\nerror:\n" ++ show e)
  Right p0 -> unless (p == p0)
              $ error ("parse error:\n" ++ intercalate "\n" (map show p)
                       ++ "\n!=\n" ++ intercalate "\n" (map show p0))

wrongCheck :: String -> IO ()
wrongCheck p | not wrongCheck' = error ("missing parse error in: " ++ p)
             | otherwise = return ()
  where wrongCheck' = case parseString p of
          Left _ -> True
          Right _ -> False

rights :: [(String, Program)]
rights = [
  -- precedence
```

```
  ("a->(0,0) || b->(0,0) @ A",
   [Com $ Par (Move ["a"] (Abs (Const 0) (Const 0)))
    (At (Move ["b"] (Abs (Const 0) (Const 0))) "A")]
  )
  ,
  -- multi
  ("viewdef A 400 -4 ((400 + 2)) \ncircle tau 1 1 (2 - 1 + 44) green \n\
\rectangle beta tau.x tau.y tau.y + 4 tau.x + 3 + 1 orange \ntau beta -> (0, 0)",
   [ Def $ Viewdef "A" (Minus (Const 400) (Const 4)) (Plus (Const 400) (Const 2))
   , Def $ Circle "tau" (Const 1) (Const 1)
     (Plus (Minus (Const 2) (Const 1)) (Const 44)) Green
   , Def $ Rectangle "beta" (Xproj "tau") (Yproj "tau")
     (Plus (Yproj "tau") (Const 4))
     (Plus (Plus (Xproj "tau") (Const 3)) (Const 1)) Orange
   , Com $ Move ["tau", "beta"] $ Abs (Const 0) (Const 0)
   ]
  )
  ]

wrongs :: [String]
wrongs = [
  -- simply wrong
  "meh"
  ,
  -- missing a space character
  "viewA"
  ,
  -- group definition without end
  "group Bouta [ Arr   \nBrr"
  ,
  -- premature bracket command end
  "{a -> (3,3  ) ||} b -> (4,4+1)"
  ,
  -- wrong Expr operator
  "circle hej 2  + 5 (1 * 88) muh.x - 14 blue"
  ,
  -- real instead of integer
  " a -> +(3.4, 0)"
  ,
  -- missing separator comma
  " baa -> (2 22)\n"
  ,
  -- wrong colour
  " rectangle\tmeat 0 0 0 0 pink"
  ,
  -- using a reserved word
  "circle circle 3 3 3 blue"
  ]



--------------------------------------------------------------------------
--- Command line entry point
--------------------------------------------------------------------------

runMain :: IO ()
runMain = do
```

```
  putStrLn "Running parser tests."
  forM_ rights rightCheck
  forM_ wrongs wrongCheck
  putStrLn "All manual parser tests pass."
  putStrLn "Testing: semantic equality"
  quickCheckWith (stdArgs { maxSuccess = 1000 }) propSemanticallyEqual
```

SalsaInterpTests.hs

```
{-# OPTIONS_GHC -Wall -fno-warn-unused-do-bind #-}
{-------------------------------------------------------------------------
Salsa interpreter tests.
-------------------------------------------------------------------------
Student name: Niels G. W. Serup <ngws@metanohi.name>
Student KU-id: njf941
-------------------------------------------------------------------------}
module SalsaInterpTests where

import Test.QuickCheck
import Control.Applicative hiding (Const)
import Data.List
import Control.Monad
import Data.Maybe (fromJust, mapMaybe)
import qualified Data.Set as S

import Misc
import Gpx
import SalsaAst
import SalsaInterp
import SalsaParser
import SalsaGenerator


-------------------------------------------------------------------------
--- QuickCheck testing
-------------------------------------------------------------------------

quickCheckTest :: (String, Property) -> IO ()
quickCheckTest (name, prop) = do
  putStrLn ("Testing " ++ name)
  r <- quickCheckWithResult (stdArgs { maxSuccess = 1000 }) prop
  case r of
    Success {} -> return ()
    _ -> error "quickCheck failed"


-- Check interpolate.
propInterpolate :: Property
propInterpolate = forAll t f
  where f (n, p0, p1) = fromIntegral (length xs) == n
                        && (sort xs == xs || sort xs == reverse xs)
                        && (n == 0 || p1 == last xs)
          where xs = interpolate n p0 p1
        t :: Gen (Integer, Position, Position)
        t = (,,) <$> choose (0, 70) <*> arbitraryPosition
            <*> arbitraryPosition
```

```
-- The number of frames must be equal to the number of non-parallel moves times
-- the framerate.
propFps :: Property
propFps = forAll t f
  where f (fps, p) = nFrames (runProg fps p)
                     == fromIntegral fps * numberOfTopCommands p
        t :: Gen (Integer, Program)
        t = (,) <$> choose (1, 68) <*> arbitraryProgram

numberOfTopCommands :: Program -> Int
numberOfTopCommands = length . filter isCom
  where isCom (Com _) = True
        isCom _ = False


-- All views of an AST must also be in the animation.
propViews :: Property
propViews = forAll arbitraryProgram $ \p ->
  sort (progViews p) == sort (animViews (runProg 1 p))

progViews :: Program -> [Ident]
progViews = mapMaybe progViews'
  where progViews' (Def (Viewdef i _ _)) = Just i
        progViews' _ = Nothing

animViews :: Animation -> [Ident]
animViews (vs, _) = map (\(i, _, _) -> i) vs


-- When a shape occurs in the AST, it must also occur in at least one frame in
-- the animation (because the program generator makes sure to move all shapes at
-- least once).
propShapeOccurence :: Property
propShapeOccurence = forAll arbitraryProgramNoProj f
  where f p = S.fromList (progShapes p) == S.fromList (animShapes (runProg 1 p))

data SShape = SRect Integer Integer String
            | SCirc Integer String
            deriving (Show, Eq, Ord)

progShapes :: Program -> [SShape]
progShapes = mapMaybe progShapes'
  where progShapes' dc = case dc of
          Def (Rectangle _ _ _ w h col) ->
            Just $ SRect (fromJust $ tryEvalExpr w)
            (fromJust $ tryEvalExpr h) $ getColourName col
          Def (Circle _ _ _ r col) ->
            Just $ SCirc (fromJust $ tryEvalExpr r) $ getColourName col
          _ -> Nothing

animShapes :: Animation -> [SShape]
animShapes = mapMaybe animShapes' . concat . snd
  where animShapes' instr = case instr of
          DrawRect _ _ w h _ col -> Just $ SRect w h col
          DrawCirc _ _ r _ col -> Just $ SCirc r col
```

```
-- When a move-to-position is absolute, there must be at least one frame in
-- which a shape is at that position.
propAbsPos :: Property
propAbsPos = forAll arbitraryProgramNoProj f
  where f p = all good $ absPoss p
          where good (e0, e1) = animHasPos (fromJust $ tryEvalExpr e0,
                                            fromJust $ tryEvalExpr e1)
                                $ runProg 1 p

absPoss :: Program -> [(Expr, Expr)]
absPoss = mapMaybe absPoss'
  where absPoss' (Com (Move _ (Abs a b))) = Just (a, b)
        absPoss' _ = Nothing

animHasPos :: Position -> Animation -> Bool
animHasPos (x, y) (_, frms) = any hasPos $ concat frms
  where hasPos instr = case instr of
          DrawRect x0 y0 _ _ _ _ -> x0 == x && y0 == y
          DrawCirc x0 y0 _ _ _ -> x0 == x && y0 == y


qcProps :: [(String, Property)]
qcProps = [ ("interpolate", propInterpolate)
          , ("fps", propFps)
          , ("views", propViews)
          , ("shape", propShapeOccurence)
          , ("abs", propAbsPos)
          ]


--------------------------------------------------------------------------
--- Manual tests
--------------------------------------------------------------------------

type ManualTest = (Integer, String, Animation)

progCheck :: ManualTest -> IO ()
progCheck (fps, s, a) =
  when (asort a' /= asort a) $
  error ("program does not create expected animation:\n"
         ++ "program:\n" ++ intercalate "\n" (map show p) ++ "\n"
         ++ "expected animation:\n" ++ fmtAni a ++ "\n"
         ++ "actual animation:\n" ++ fmtAni a'
        )
  where a' = runProg fps p
        p = either (\e -> error ("program did not parse:\n"
                                 ++ show e ++ "\nprogram:\n" ++ s))
            id $ parseString s
        fmtAni ani = show (fst ani) ++ "\n" ++ intercalate "\n"
                     (map show (snd ani))
        asort (x, y) = (sort x, map sort y)


-- Simple, small test to see if the very basics work.
testSimple :: ManualTest
testSimple = (
```

```
  1,
  "viewdef Mu 100 100\nrectangle x 10 10 10 10 plum\tx->(10,10)",
  ([("Mu", 100, 100)],
   [ [ DrawRect 10 10 10 10 "Mu" "plum"
     ]
   ])
  )


-- Animation interpolating.
testAnimateInterpolate :: ManualTest
testAnimateInterpolate = (
  4,
  "viewdef Tt 300 333\n\
\circle circa 10 90 20 blue\n\
\circa -> (10, 10)",
  ([("Tt", 300, 333)],
   [ [ DrawCirc 10 70 20 "Tt" "blue"
     ]
   , [ DrawCirc 10 50 20 "Tt" "blue"
     ]
   , [ DrawCirc 10 30 20 "Tt" "blue"
     ]
   , [ DrawCirc 10 10 20 "Tt" "blue"
     ]
   ])
  )


-- Shapes
testShapes :: ManualTest
testShapes = (
  1,
  "viewdef Niels 1338 1336\ncircle c 1 2 3 blue\
\\nrectangle r 1 2 3 4 plum\nc r->+(0,0)",
  ([("Niels", 1338, 1336)],
   [ [ DrawCirc 1 2 3 "Niels" "blue"
     , DrawRect 1 2 3 4 "Niels" "plum"
     ]
   ])
  )


-- Multiple moves.
testMultipleMoves :: ManualTest
testMultipleMoves = (
  1,
  "viewdef A 1 1 circle x 1 1 1 orange x -> (2, 2) x -> +(1, 1)",
  ([("A", 1, 1)],
   [ [ DrawCirc 2 2 1 "A" "orange"
     ]
   , [ DrawCirc 3 3 1 "A" "orange"
     ]
   ])
  )
```

```
-- Groups.
testGroups :: ManualTest
testGroups = (
  1,
  "viewdef A 3 3\n\
\viewdef B 5 5\n\
\viewdef Otto 999 991\n\
\group Brandenburg [Otto A B]\n\
\view Brandenburg\n\
\circle x 0 0 25 green\n\
\view B\n\
\x -> (1, 10)\n\
\view Brandenburg\n\
\x->+(5,5)",
  ([("Otto", 999, 991), ("B", 5, 5), ("A", 3, 3)],
   [ [ DrawCirc 0 0 25 "Otto" "green"
     , DrawCirc 0 0 25 "A" "green"
     , DrawCirc 1 10 25 "B" "green"
     ]
   , [ DrawCirc 5 5 25 "Otto" "green"
     , DrawCirc 5 5 25 "A" "green"
     , DrawCirc 6 15 25 "B" "green"
     ]
   ])
  )


-- Use of @.
testAt :: ManualTest
testAt = (
  1,
  "viewdef I 10 10 circle a 1 1 1 red viewdef Y 2 2\n a->(0,0) @ I",
  ([("I", 10, 10), ("Y", 2, 2)],
   [ [ DrawCirc 0 0 1 "I" "red"
     ]
   ])
  )


-- Use of ||.
testPar :: ManualTest
testPar = (
  1,
  "viewdef Uha 100 100\t circle a 1 1 1 red\t rectangle b 1 1 1 1 red\
\\t a -> (7, 7) || b -> +(8, 1)",
  ([("Uha", 100, 100)],
   [ [ DrawCirc 7 7 1 "Uha" "red"
     , DrawRect 9 2 1 1 "Uha" "red"
     ]
   ])
  )


-- Expr.
testExpr :: ManualTest
testExpr = (
  1,
```

```
  "viewdef Thing 3 + 9-11 + 500 ( (300))\n\
\circle b 2 3 4 green\n\
\rectangle rrr 4 4 b.x + 14 (4 + b.y - b.y + b.y) red\n\
\b rrr -> +(1,0-0)",
  ([("Thing", 501, 300)],
   [ [ DrawCirc 3 3 4 "Thing" "green"
     , DrawRect 5 4 16 7 "Thing" "red"
     ]
   ])
  )



progs :: [ManualTest]
progs = [ testSimple
        , testAnimateInterpolate
        , testShapes
        , testMultipleMoves
        , testGroups
        , testAt
        , testPar
        , testExpr
        ]


-------------------------------------------------------------------------
--- Command line entry point
-------------------------------------------------------------------------

runMain :: IO ()
runMain = do
  putStrLn "Running interpreter tests."
  forM_ progs progCheck
  putStrLn "All manual interpreter tests pass."
  forM_ qcProps quickCheckTest
```

A.1.5 Scripts

animate.hs

```
#!/usr/bin/env runghc
{-# OPTIONS_GHC -Wall -fno-warn-unused-do-bind #-}
{-------------------------------------------------------------------------
Command line tool to generate animations.
-------------------------------------------------------------------------
Student name: Niels G. W. Serup <ngws@metanohi.name>
Student KU-id: njf941
-------------------------------------------------------------------------}
module Main (main) where

import Control.Monad
import System.Environment

import Salsa
```

```
main :: IO ()
main = do
  args <- getArgs
  case args of
    [path, fps] -> printAni =<< runFile (read fps) path
    [path] -> printAni =<< runFile 10 path
    _ -> putStrLn "usage: animate.hs FILEPATH [FPS (default is 10)]"
  where printAni (a, b) = do
          print a
          putStrLn ""
          if null b
            then putStrLn "[]"
            else do
            putStr "[ " >> print (head b)
            forM_ (tail b) $ \x -> putStr ", " >> print x
            putStrLn "]"
```

test.hs

```
#!/usr/bin/env runghc
{-# OPTIONS_GHC -Wall -fno-warn-unused-do-bind #-}
{-------------------------------------------------------------------
Command line tool to run tests.
-------------------------------------------------------------------
Student name: Niels G. W. Serup <ngws@metanohi.name>
Student KU-id: njf941
-------------------------------------------------------------------}
module Main (main) where

import qualified SalsaParserTests as P
import qualified SalsaInterpTests as I

main :: IO ()
main = do
  P.runMain
  I.runMain
```

A.2 AT Server

A.2.1 API

at_server.erl

```
%%%-------------------------------------------------------------------
%%% @author Niels G. W. Serup <ngws@metanohi.name>
%%% @author Michael Kirkedal Thomsen <shapper@diku.dk>
%%% @copyright (C) 2013, Michael Kirkedal Thomsen
%%% @doc
%%% Atomic Transaction Server API.
%%% @end
%%% Created: Oct 2013 by Michael Kirkedal Thomsen <shapper@diku.dk>
%%%-------------------------------------------------------------------
%%% Student name: Niels G. W. Serup <ngws@metanohi.name>
%%% Student KU-id: njf941
%%%-------------------------------------------------------------------
```

```
-module(at_server).

-export([start/1, stop/1, begin_t/1, doquery/2, query_t/3,
         update_t/3, commit_t/2]).

-type state_transformer() :: fun((any()) -> any()).


-spec start(State::any()) -> {ok, AT::pid()}.
%% @doc
%% Create an atomic transaction server ‘AT’ with ‘State’ as the initial state.
%% @end
start(State) ->
    gen_fsm:start_link(at_server_master, {State, [], 0}, []).


-spec stop(AT::pid()) -> {ok, State::any()}.
%% @doc
%% Abort all running transactions in the transaction server ‘AT’ and stop
%% itself.  Return the current ‘State’ of the server.
%% @end
stop(AT) ->
    {ok, gen_fsm:sync_send_all_state_event(AT, stop)}.


-spec doquery(AT::pid(), Fun::state_transformer())
             -> ({ok, StateNew::any()} | error).
%% @doc
%% Query the server with ‘Fun’ on its current ‘State’.  Return the result, or
%% ‘error’ if ‘Fun’ fails.
%% @end
doquery(AT, Fun) ->
    gen_fsm:sync_send_event(AT, {doquery, Fun}).


-spec begin_t(AT::pid()) -> {ok, Ref::pid()}.
%% @doc
%% Start a new atomic transaction on ‘AT’.  Return the new unique transaction
%% id.
%% @end
begin_t(AT) ->
    {ok, gen_fsm:sync_send_event(AT, fork)}.


-spec query_t(AT::pid(), Ref::pid(), Fun::state_transformer())
             -> ({ok, StateNew::any()} | aborted).
%% @doc
%% Query the helper transaction ‘Ref’ with ‘Fun’ on its current ‘State’.  Return
%% the result, or ‘aborted’ if ‘Fun’ fails (in which case the transaction is
%% also aborted), if the transaction has already been aborted, or if the
%% transaction does not exist.
%% @end
query_t(AT, Ref, Fun) ->
    aborted_if_error(fun() -> gen_fsm:sync_send_event(Ref, {doquery, Fun, AT}) end).
```

```
-spec update_t(AT::pid(), Ref::pid(), Fun::state_transformer())
              -> (ok | aborted).
%% @doc
%% Update the ‘State’ of the helper transaction ‘Ref’ with ‘Fun’.  Return
%% ‘aborted’ if the transaction does not exist, otherwise ‘ok’.
%% @end
update_t(AT, Ref, Fun) ->
    case erlang:is_process_alive(Ref) of
        true -> aborted_if_error(
                  fun() -> gen_fsm:send_event(Ref, {update, Fun, AT}) end);
        false -> aborted
    end.


-spec (commit_t(AT::pid(), Ref::pid()) -> (ok | aborted)).
%% @doc
%% Commit the state of ‘Ref’ to ‘AT’.  Return ‘ok’ if successful, otherwise
%% ‘aborted’.
%% @end
commit_t(AT, Ref) ->
    aborted_if_error(fun() -> gen_fsm:sync_send_event(Ref, {commit, AT}) end).


-spec aborted_if_error(Fun::fun(() -> any())) -> (any() | aborted).
%% @doc Returns ‘aborted’ if an error occurs.
aborted_if_error(Fun) ->
    try Fun()
    catch
        _:_ -> aborted
    end.
```

at_extapi.erl

```
%%%-------------------------------------------------------------------
%%% @author Niels G. W. Serup <ngws@metanohi.name>
%%% @author Michael Kirkedal Thomsen <shapper@diku.dk>
%%% @copyright (C) 2013, Michael Kirkedal Thomsen
%%% @doc
%%% Atomic Transaction Server Extended API.
%%% @end
%%% Created: Oct 2013 by Michael Kirkedal Thomsen <shapper@diku.dk>
%%%-------------------------------------------------------------------
%%% Student name: Niels G. W. Serup <ngws@metanohi.name>
%%% Student KU-id: njf941
%%%-------------------------------------------------------------------

-module(at_extapi).

-export([abort/2, tryUpdate/2, ensureUpdate/2, choiceUpdate/3]).


-spec abort(AT::pid(), Ref::pid()) -> aborted.
%% @doc Force an abort of ‘Ref’ on ‘AT’.
abort(AT, Ref) ->
    at_server:update_t(AT, Ref, fun() -> erlang:error(aborted) end),
    aborted.
```

```
-spec (tryUpdate(AT::pid(), Fun::at_server:state_transformer())
       -> (ok | error | aborted)).
%% @doc
%% Try to update the ‘State’ on ‘AT’ with ‘Fun’.  Return ‘ok’ on success,
%% ‘error’ if ‘Fun’ fails, and ‘aborted’ if another transaction committed before
%% this one did.
%% @end
tryUpdate(AT, Fun) ->
    {ok, Ref} = at_server:begin_t(AT),
    at_server:update_t(AT, Ref, Fun),
    at_server:commit_t(AT, Ref).


-spec (ensureUpdate(AT::pid(), Fun::at_server:state_transformer())
       -> {ok, error}).
%% @doc
%% Has the same ‘ok’ and ‘error’ effects as ‘tryUpdate’, but always comes
%% through with its commit (so it never returns ‘aborted’).
%% @end
%% @see tryUpdate
ensureUpdate(AT, Fun) ->
    Res = tryUpdate(AT, Fun),
    case Res of
        aborted -> ensureUpdate(AT, Fun);
        T -> T
    end.


-spec (choiceUpdate(AT::pid(),
                    Fun::fun((State::any(), E::any())
                             -> NewState::any()), [any()]) -> (FinalState::any() | error)).
%% Commit and return the ‘NewState’ whose ‘Fun’ finishes first.  Abort the rest.
%% However, if all ‘Fun’ applications return ‘error’, then return ‘error’.
choiceUpdate(AT, Fun, Elements) ->
    S = self(),
    %% The actual core function might have some spawned processes sending data
    %% back before they are killed, but we don’t want that data ending up in
    %% this process, so we start a new one.  This could most likely be done
    %% better.
    spawn(fun() -> choiceUpdate1(AT, Fun, Elements, S) end),
    receive V -> V end.

choiceUpdate1(AT, Fun, Elements, ReplyTo) ->
    {ok, S} = at_server:doquery(AT, fun at_misc:id/1),
    Self = self(),
    Pids = lists:map(fun(E) -> spawn(fun() -> Self ! try Fun(S, E)
                                                     of V -> {ok, V}
                                                     catch _:_ -> error
                                                     end
                                     end) end, Elements),
    Res = choiceUpdateRec(length(Elements)),
    lists:map(fun(P) -> exit(P, kill) end, Pids),
    case Res =/= error of
        true -> ensureUpdate(AT, fun(_) -> Res end);
        false -> ok
    end,
```

```
    ReplyTo ! Res.

choiceUpdateRec(M) ->
    receive
        {ok, V} -> V;
        error -> case M > 1 of
                     true -> choiceUpdateRec(M - 1);
                     false -> error
                 end
    end.
```

A.2.2 Implementation

at_server_master.erl

```
%%%-------------------------------------------------------------------
%%% @author Niels G. W. Serup <ngws@metanohi.name>
%%% @doc
%%% Implementation of the atomic transaction server.
%%% @end
%%% Created: Nov 2013 by Niels G. W. Serup <ngws@metanohi.name>
%%%-------------------------------------------------------------------
%%% Student name: Niels G. W. Serup <ngws@metanohi.name>
%%% Student KU-id: njf941
%%%-------------------------------------------------------------------

-module(at_server_master).
-behaviour(gen_fsm).


-export([init/1, handle_event/3, terminate/3, handle_info/3,
         code_change/4, handle_sync_event/4]).
-export([ready/3]).


%%%-------------------------------------------------------------------
%%% Unused gen_fsm callbacks.
%%%-------------------------------------------------------------------

handle_info(_Info, StateName, State) ->
    {next_state, StateName, State}.

terminate(_Reason, _StateName, _State) ->
    ok.

code_change(_OldVsn, StateName, State, _Extra) ->
    {ok, StateName, State}.

handle_event(_Event, _StateName, State) ->
    {stop, exit, State}.


%%%-------------------------------------------------------------------
%%% Used gen_fsm callbacks and helpers.
%%%-------------------------------------------------------------------
```

```
-type master_state() :: {S::any(), Helpers::[pid()], Iter::integer()}.


init(State) ->
    {ok, ready, State}.


%% @doc Abort all transactions and stop.
handle_sync_event(stop, _From, _StateName, State = {S, Helpers, _Iter}) ->
    abort_helpers(Helpers),
    {stop, normal, S, State}.


%% @doc Abort all transactions.
abort_helpers(Helpers) ->
    lists:map(fun(H) -> at_extapi:abort(self(), H) end, Helpers).


%% @doc Query, fork, or commit.
-spec (ready(Action::any(), From::pid(), State::master_state())
       -> NewAction::any()).
ready(Action, _From, State) ->
    {Reply, NextState} =
        case Action of
            {doquery, Fun} -> doquery(State, Fun);
            fork -> fork(State);
            {commit, S1, HelperIter} -> commit(State, S1, HelperIter)
        end,
    {reply, Reply, ready, NextState}.


-spec doquery(State::master_state(), Fun::at_server:state_transformer())
             -> {({ok, StateNew::master_state()} | error), master_state()}.
%% @doc Query the server.
doquery(State = {S, _Helpers, _Iter}, Fun) ->
    R = try Fun(S) of
            S1 -> {ok, S1}
        catch
            _:_ -> error
        end,
    {R, State}.


-spec fork(State::master_state()) -> {Ref::pid(), StateNew::master_state()}.
%% @doc Create a new helper transaction.
fork({S, Helpers, Iter}) ->
    {ok, Ref} = gen_fsm:start_link(at_server_helper, {S, self(), Iter}, []),
    {Ref, {S, [Ref | Helpers], Iter}}.


-spec commit(State::master_state(), S1::any(), HelperIter::integer())
            -> {ok, NewState::master_state()}.
%% @doc Try to commit a result from a helper transaction.
commit(State = {_S, Helpers, Iter}, S1, HelperIter) ->
    case HelperIter =:= Iter of
        true ->
            abort_helpers(Helpers),
```

```
            {ok, {S1, [], Iter + 1}};
        false ->
            {aborted, State}
    end.
```

at_server_helper.erl

```
%%%-------------------------------------------------------------------
%%% @author Niels G. W. Serup <ngws@metanohi.name>
%%% @doc
%%% Implementation of the atomic transaction helper server.
%%% @end
%%% Created: Nov 2013 by Niels G. W. Serup <ngws@metanohi.name>
%%%-------------------------------------------------------------------
%%% Student name: Niels G. W. Serup <ngws@metanohi.name>
%%% Student KU-id: njf941
%%%-------------------------------------------------------------------

-module(at_server_helper).
-behaviour(gen_fsm).


-export([init/1, handle_event/3, terminate/3, handle_info/3,
         code_change/4, handle_sync_event/4]).
-export([ready/2, ready/3]).


%%%-------------------------------------------------------------------
%%% Unused gen_fsm callbacks.
%%%-------------------------------------------------------------------

handle_info(_Info, StateName, State) ->
    {next_state, StateName, State}.

terminate(_Reason, _StateName, _State) ->
    ok.

code_change(_OldVsn, StateName, State, _Extra) ->
    {ok, StateName, State}.

handle_sync_event(_Event, _From, _StateName, State) ->
    {stop, exit, State}.


%%%-------------------------------------------------------------------
%%% Used gen_fsm callbacks and helpers.
%%%-------------------------------------------------------------------

-type helper_state() :: {S::any(), AT::pid(), Iter::integer()}.


init(State) ->
    {ok, ready, State}.


%% @doc Stop.
handle_event(stop, _From, State) ->
```

```
    {stop, normal, State}.


%% @doc Update.
ready(Action, State) ->
    case Action of
        {update, Fun, AT} -> update(State, Fun, AT)
    end.


%% @doc Query or commit.
ready(Action, _From, State) ->
    case Action of
        {doquery, Fun, AT} -> doquery(State, Fun, AT);
        {commit, AT} -> commit(State, AT)
    end.


-spec (update(State::helper_state(), Fun::at_server:state_transformer(),
              AT1::pid()) -> ({stop, term(), helper_state()}
                              | {next_state, ready, helper_state()})).
%% @doc Update the helper transaction.
update(State = {S, AT, Iter}, Fun, AT1) ->
    case AT =/= AT1 of
        true ->
            {stop, normal, State};
        false ->
            try Fun(S) of
                S1 -> {next_state, ready, {S1, AT, Iter}}
            catch
                _:_ -> {stop, normal, S}
            end
    end.


-spec (doquery(State::helper_state(), Fun::at_server:state_transformer(),
               AT1::pid()) -> ({stop, term(), any(), helper_state()}
                               | {reply, any(), ready, helper_state()})).
%% @doc Query the helper transaction.
doquery(State = {S, AT, _Iter}, Fun, AT1) ->
    case AT =/= AT1 of
        true ->
            {stop, normal, aborted, State};
        false ->
            try Fun(S) of
                S1 -> {reply, {ok, S1}, ready, State}
            catch
                _:_ -> {stop, normal, aborted, State}
            end
    end.


-spec (commit(State::helper_state(), AT1::pid())
       -> ({stop, term(), any(), helper_state()}
           | {reply, any(), ready, helper_state()})).
%% @doc Commit to the master.
commit(State = {S, AT, Iter}, AT1) ->
```

```
    case AT =/= AT1 of
        true ->
            {stop, normal, aborted, State};
        false ->
            R = gen_fsm:sync_send_event(AT, {commit, S, Iter}),
            {stop, normal, R, State}
    end.
```

A.2.3 Tests

at_server_tests.erl

```
%%%-------------------------------------------------------------------
%%% @doc
%%% Atomic Transaction Server Tests.
%%% @end
%%%-------------------------------------------------------------------
%%% Student name: Niels G. W. Serup
%%% Student KU-id: njf941
%%%-------------------------------------------------------------------

-module(at_server_tests).
-include_lib("eunit/include/eunit.hrl").

%% Test that basic function application works on the server.
function_application_test() ->
    A = random:uniform(4442),
    B = random:uniform(3342),
    {ok, AT} = at_server:start(A),
    M = A + B,
    {ok, M} = at_server:doquery(AT, fun(N) -> N + B end),
    error = at_server:doquery(AT, fun(_) -> throw(noway) end),
    {ok, A} = at_server:stop(AT).


%% Test that sequential transactions work.
sequential_transactions_test() ->
    {ok, AT} = at_server:start([3, 1]),

    {ok, Ref0} = at_server:begin_t(AT),
    {ok, [1]} = at_server:query_t(AT, Ref0, fun([_ | Xs]) -> Xs end),
    ok = at_server:update_t(AT, Ref0, fun(Xs) -> [ahem | Xs] end),
    {ok, [ahem, 3, 1]} = at_server:query_t(AT, Ref0, fun at_misc:id/1),
    {ok, [3, 1]} = at_server:doquery(AT, fun at_misc:id/1),
    ok = at_server:commit_t(AT, Ref0),
    aborted = at_server:query_t(Ref0, Ref0, fun at_misc:id/1),
    {ok, [ahem, 3, 1]} = at_server:doquery(AT, fun at_misc:id/1),
    aborted = at_server:query_t(AT, Ref0, fun at_misc:id/1),
    aborted = at_server:commit_t(AT, Ref0),
    aborted = at_server:update_t(AT, Ref0, fun at_misc:id/1),

    {ok, Ref1} = at_server:begin_t(AT),
    {ok, [ahem, 3, 1]} = at_server:query_t(AT, Ref1, fun at_misc:id/1),
    ok = at_server:update_t(AT, Ref1, fun(Xs) -> [ahem | Xs] end),
    {ok, [ahem, ahem, 3, 1]} = at_server:query_t(AT, Ref1, fun at_misc:id/1),
    {ok, [ahem, 3, 1]} = at_server:doquery(AT, fun at_misc:id/1),
```

```
    ok = at_server:update_t(AT, Ref1, fun(_) -> muuuuh end),
    ok = at_server:commit_t(AT, Ref1),
    aborted = at_server:commit_t(AT, Ref1),
    {ok, muuuuh} = at_server:doquery(AT, fun at_misc:id/1),
    aborted = at_server:update_t(AT, Ref1, fun at_misc:id/1),
    aborted = at_server:query_t(AT, Ref1, fun at_misc:id/1),

    aborted = at_server:update_t(AT, Ref0, fun at_misc:id/1),
    aborted = at_server:query_t(AT, Ref0, fun at_misc:id/1),
    aborted = at_server:commit_t(AT, Ref0),

    {ok, muuuuh} = at_server:stop(AT).


%% Test that concurrent transactions seem to work.
concurrent_transactions_test() ->
    {ok, AT} = at_server:start(start),
    {ok, Ref0} = at_server:begin_t(AT),
    {ok, Ref1} = at_server:begin_t(AT),
    {{NRef0, T0}, {NRef1, T1}} =
        case random:uniform(2) =:= 1 of
            true -> {{Ref0, ref0}, {Ref1, ref1}};
            false -> {{Ref1, ref1}, {Ref0, ref0}}
        end,
    S = self(),
    spawn(fun() ->
                  at_server:update_t(AT, NRef0, fun(_) -> T0 end),
                  S ! {T0, at_server:commit_t(AT, NRef0)}
          end),
    spawn(fun() ->
                  at_server:update_t(AT, NRef1, fun(_) -> T1 end),
                  S ! {T1, at_server:commit_t(AT, NRef1)}
          end),
    {RA, SA} = receive T -> T end,
    {RB, SB} = receive U -> U end,

    ?assert((SA =:= ok andalso SB =:= aborted)
            orelse (SB =:= ok andalso SA =:= aborted)),
    AW =
        case SA =:= ok of
            true -> RA;
            false -> RB
        end,
    aborted = at_server:commit_t(AT, Ref0),
    aborted = at_server:update_t(AT, Ref0, fun at_misc:id/1),
    aborted = at_server:query_t(AT, Ref1, fun at_misc:id/1),
    aborted = at_server:commit_t(AT, Ref1),

    {ok, AW} = at_server:doquery(AT, fun at_misc:id/1),
    {ok, AW} = at_server:stop(AT).


%% Test tryUpdate.
tryUpdate_test() ->
    {ok, AT} = at_server:start(start),
    S = self(),
    spawn(fun() -> aborted = at_extapi:tryUpdate(
```

```
                                AT, fun(_) -> S ! hullo,
                                              timer:sleep(100),
                                              ref0
                                    end),
                        S ! at_server:stop(AT)
              end),
    receive hullo -> ok end,
    ok = at_extapi:tryUpdate(AT, fun(_) -> ref1 end),
    receive {ok, ref1} -> ok end.


%% Test ensureUpdate.
ensureUpdate_test() ->
    {ok, AT} = at_server:start(start),
    Xs = lists:seq(1, 100),
    S = self(),
    lists:map(fun(X) -> spawn(
                           fun() -> S ! at_extapi:ensureUpdate(
                                          AT, fun(_) -> X end) end) end, Xs),
    As = lists:map(fun(_) -> ok end, Xs),
    Bs = lists:map(fun(_) -> receive T -> T end end, Xs),
    As = Bs,
    {ok, N} = at_server:doquery(AT, fun at_misc:id/1),
    ?assert(lists:member(N, Xs)),
    at_server:stop(AT).


%% Test choiceUpdate.
choiceUpdate_test() ->
    State = 5,
    {ok, AT} = at_server:start(State),
    Fun = fun(S, E) -> S + E end,
    Elements = lists:seq(1, 336),
    Res = at_extapi:choiceUpdate(AT, Fun, Elements),
    ?assert(lists:member(Res, lists:map(fun(E) -> Fun(State, E) end, Elements))).
```